# Spark源码之Standalone模式下master持久化引擎讲解

2017-06-09 浪尖 [Spark高级玩法](https://mp.weixin.qq.com/s?__biz=MzA3MDY0NTMxOQ==&mid=2247483685&idx=1&sn=a93db24d5eec1b4093e0042db9589b69&chksm=9f38e20da84f6b1bfcc4fcb960f17099d379c0143767f7a7495d294f01e1905ab482b89882df&scene=21##)

Standalone 模式下Master为了保证故障恢复，会持久化一些重要的数据，来避免master故障导致集群不可用这种情况(也即单点故障)。目前，有四种持久化策略:

1,基于zookeeper的持久化引擎。

2,基于文件的持久化引擎。

3,用户自定义持久化引擎。

4,不使用持久化引擎。

**在master的OnStart方法中，对应的源码如下:**

**val** *serializer* = **new** JavaSerializer(conf)

**val** (*persistenceEngine\_*, leaderElectionAgent\_) = RECOVERY\_MODE **match** {

**case "ZOOKEEPER"** =>

logInfo(**"Persisting recovery state to ZooKeeper"**)

**val** zkFactory =

**new** ZooKeeperRecoveryModeFactory(conf, *serializer*)

(zkFactory.createPersistenceEngine(), zkFactory.createLeaderElectionAgent(**this**))

**case "FILESYSTEM"** =>

**val** fsFactory =

**new** FileSystemRecoveryModeFactory(conf, *serializer*)

(fsFactory.createPersistenceEngine(), fsFactory.createLeaderElectionAgent(**this**))

**case "CUSTOM"** =>

**val** clazz = Utils.classForName(conf.get(**"spark.deploy.recoveryMode.factory"**))

**val** factory = clazz.getConstructor(*classOf*[SparkConf], *classOf*[Serializer])

.newInstance(conf, *serializer*)

.asInstanceOf[StandaloneRecoveryModeFactory]

(factory.createPersistenceEngine(), factory.createLeaderElectionAgent(**this**))

**case** \_ =>

(**new** BlackHolePersistenceEngine(), **new** MonarchyLeaderAgent(**this**))

}

persistenceEngine = *persistenceEngine\_*

leaderElectionAgent = leaderElectionAgent\_

默认，情况下是无持久化引擎，也就是没有ha策略。Spark提供的可用的ha策略：基于文件系统的和基于zookeeper。配置方法如下:

**基于文件系统：**

|  |  |
| --- | --- |
| property | Meaning |
| spark.deploy.recoveryMode | FILESYSTEM |
| spark.deploy.recoveryDirectory | 用来恢复状态的目录 |

**基于zookeeper:**

|  |  |
| --- | --- |
| property | Meaning |
| spark.deploy.recoveryMode | ZOOKEEPER |
| spark.deploy.zookeeper.url | e.g., 192.168.1.100:2181,192.168.1.101:2181 |
| spark.deploy.zookeeper.dir | zookeeper保存恢复状态的目录 |

生产环境中可用的是基于zookeeper的持久化引擎。

基于zookeeper持久化策略，会允许我们同时运行多个master，然后支持leader选举，最终是一个leader，其余是standby。

**Spark的Master的leader选举实现**

Spark源码里面使用的是CuratorFramework，跟zookeeper交流。该框架有以下特点:

1，自动连接管理：自动处理zookeeper的连接和重试存在一些潜在的问题；可以watch NodeDataChanged event和获取updateServerList;Watches可以自动被Cruator recipes删除；

2，更加简洁的API：简化raw zookeeper方法，事件等；提供现代流式API接口

3，Recipe实现：leader选举，分布式锁，path缓存，和watcher,分布式队列，Barriers等。

**Spark源码里面使用了LeaderLatch实现选举功能。**这个实现实际是基于zookeeper的节点类型来做，zookeeper有四种节点类型:

1，持久节点（PERSISTENT）

节点创建后，会一直存在，不会因客户端会话失效而删除；

2，持久顺序节点（PERSISTENT\_SEQUENTIAL）

基本特性与持久节点一致，创建节点的过程中，zookeeper会在其名字后自动追加一个单调增长的数字后缀，作为新的节点名；

3，临时节点（EPHEMERAL）

客户端会话失效或连接关闭后，该节点会被自动删除，且不能再临时节点下面创建子节点。

4，临时顺序节点（EPHEMERAL\_SEQUENTIAL）

基本特性与临时节点一致，创建节点的过程中，zookeeper会在其名字后自动追加一个单调增长的数字后缀，作为新的节点名；

**LeaderLatch实现leader选举实际上基于临时顺序节点来做的。**

**Spark源码里面基于zookeeper的leader选举具体实现过程源码如下:**

在master的OnStart方法里面

leaderElectionAgent = *leaderElectionAgent\_*

实际是在构建zookeeper的持久化引擎的时候，构建的

(zkFactory.createPersistenceEngine(), zkFactory.createLeaderElectionAgent(**this**))

在createLeaderElectionAgent方法里面构建了

**new** ZooKeeperLeaderElectionAgent(master, conf)

该对象，继承了LeaderLatchListener，并且覆盖了notLeader和isLeader两个重要的方法具体。

在ZooKeeperLeaderElectionAgent构建的时候调用了自己的start方法，该方法构建了LeaderLatch，并添加ZooKeeperLeaderElectionAgent作为其listener。

**private def** start() {

logInfo(**"Starting ZooKeeper LeaderElection agent"**)

*zk* = SparkCuratorUtil.*newClient*(conf)

*leaderLatch* = **new** LeaderLatch(*zk*, *WORKING\_DIR*)

*leaderLatch*.addListener(**this**)

*leaderLatch*.start()

}

Leader选举在zookeeper的临时节点的路径为

**val** *WORKING\_DIR* = conf.get(**"spark.deploy.zookeeper.dir"**, **"/spark"**) + **"/leader\_election"**

执行ZooKeeperLeaderElectionAgent对象的start方法之后，每当该对象所在的master由standby变为Leader的时候，会调用isLeader()方法。由Leader变为StandBy的时候会调用notLeader()。我们就可以在这两个方法里实现自己要的状态切换的相关操作。

**override def** isLeader() {

synchronized {

*// could have lost leadership by now.*

**if** (!*leaderLatch*.hasLeadership) {

**return**

}

logInfo(**"We have gained leadership"**)

updateLeadershipStatus(**true**)

}

}

**override def** notLeader() {

synchronized {

*// could have gained leadership by now.*

**if** (*leaderLatch*.hasLeadership) {

**return**

}

logInfo(**"We have lost leadership"**)

updateLeadershipStatus(**false**)

}

}

要实现，我们自己应用的ha，也可基于此方法。

# spark源码系列之内部通讯的三种机制

原创 2017-07-09 浪尖 [Spark高级玩法](https://mp.weixin.qq.com/s?__biz=MzA3MDY0NTMxOQ==&mid=2247483733&idx=1&sn=92e7bcf399a8aaa24f7b6c37b98b3b2e&chksm=9f38e27da84f6b6bcd9e26db1f11b83fd4dca9d4e3512a25fd3b4eae012065c939d6ed6ecaad&scene=21##)

本文是以spark1.6.0的源码为例讲解。

Spark为协调各个组件完成任务及内部任务处理采用了多种方式进行了各个组件之间的通讯。总共三个部分牵涉的功能是：

1，DAG相关的DAGSchedulerEventProcessLoop。

2，sparkUI相关的SparkListener

3，RPC相关netty RPC流程。本文只讲流程，后面会详细介绍。

**一，单个部件自己消息处理方式**

DAGSchedulerEventProcessLoop该类继承自EventLoop。是一个典型的生产消费模型。

A)生产者

通过调用

DAGSchedulerEventProcessLoop.post(event: E)

来将消息进行发布。

B)消费者

Eventloop内部维护了一个线程，循环的消费消息eventQueue.take()，调用onReceive(event)进行处理。DAGSchedulerEventProcessLoop内部实现了doOnReceive，对事件进行模式匹配然后交给具体的消息处理函数。

**private def** doOnReceive(event: DAGSchedulerEvent): Unit = event **match** {

**case** JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) =>

dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener,properties)

**case** MapStageSubmitted(jobId, dependency, callSite, listener, properties) =>

dagScheduler.handleMapStageSubmitted(jobId, dependency, callSite, listener, properties)

**case** StageCancelled(stageId) =>

dagScheduler.handleStageCancellation(stageId)

**case** JobCancelled(jobId) =>

dagScheduler.handleJobCancellation(jobId)

**case** JobGroupCancelled(groupId) =>

dagScheduler.handleJobGroupCancelled(groupId)

**case** AllJobsCancelled =>

dagScheduler.doCancelAllJobs()

**case** ExecutorAdded(execId, host) =>

dagScheduler.handleExecutorAdded(execId, host)

**case** ExecutorLost(execId) =>

dagScheduler.handleExecutorLost(execId, fetchFailed = **false**)

**case** BeginEvent(task, taskInfo) =>

dagScheduler.handleBeginEvent(task, taskInfo)

**case** GettingResultEvent(taskInfo) =>

dagScheduler.handleGetTaskResult(taskInfo)

**case** completion @ CompletionEvent(task, reason, \_, \_, taskInfo, taskMetrics) =>

dagScheduler.handleTaskCompletion(completion)

**case** TaskSetFailed(taskSet, reason, exception) =>

dagScheduler.handleTaskSetFailed(taskSet, reason, exception)

**case** ResubmitFailedStages =>

dagScheduler.resubmitFailedStages()

}

C),消息缓存

消息最终是存储于EventLoop的new LinkedBlockingDeque[E]()里。

**二，SparkListeners和ListenerBus**

SparkUI的各个监控指标都是，由ListenerBus最为生产者将消息，推送到消息缓存出默认支持1万，然后推送给各个Listener进行处理，然后我们的Spark的webUIPage去获取各个Listener的数据，进行展示。

A)，生产者

LiveListenerBus/StreamingListenerBus调用其父类AsynchronousListenerBus的post方法将消息加入new LinkedBlockingQueue[E](EVENT\_QUEUE\_CAPACITY)，容量1万。

**val** *eventAdded* = eventQueue.offer(event)

B),消费者

AsynchronousListenerBus内部维护了一个消费者线程，线程内部有while(true)进行消息处理。

**val** *event* = eventQueue.poll  
postToAll(*event*)

C),消息的具体处理

ListenerBus的postToAll方法，会遍历所有注册了的Listener。

**final def** postToAll(event: E): Unit = {  
 **val** iter = listeners.iterator  
 **while** (iter.hasNext) {  
 **val** listener = iter.next()  
 **try** {  
 onPostEvent(listener, event)  
 } **catch** {  
 **case** NonFatal(e) =>  
 logError(**s"Listener $**{Utils.getFormattedClassName(listener)} **threw an exception"**, e)  
 }  
 }  
}

最终在onPostEvent方法中将消息进行了处理。onPostEvent在源码中的两个重要现:

SparkListenerBus和StreamingListenerBus内部的onPostEvent。

**private**[spark] **trait** SparkListenerBus **extends** ListenerBus[SparkListener, SparkListenerEvent] {  
  
 **override def** onPostEvent(listener: SparkListener, event: SparkListenerEvent): Unit = {  
 event **match** {  
 **case** stageSubmitted: SparkListenerStageSubmitted =>  
 listener.onStageSubmitted(stageSubmitted)  
 **case** stageCompleted: SparkListenerStageCompleted =>  
 listener.onStageCompleted(stageCompleted)  
 **case** jobStart: SparkListenerJobStart =>  
 listener.onJobStart(jobStart)  
 **case** jobEnd: SparkListenerJobEnd =>  
 listener.onJobEnd(jobEnd)  
 **case** taskStart: SparkListenerTaskStart =>  
 listener.onTaskStart(taskStart)  
 **case** taskGettingResult: SparkListenerTaskGettingResult =>  
 listener.onTaskGettingResult(taskGettingResult)  
 **case** taskEnd: SparkListenerTaskEnd =>  
 listener.onTaskEnd(taskEnd)  
 **case** environmentUpdate: SparkListenerEnvironmentUpdate =>  
 listener.onEnvironmentUpdate(environmentUpdate)  
 **case** blockManagerAdded: SparkListenerBlockManagerAdded =>  
 listener.onBlockManagerAdded(blockManagerAdded)  
 **case** blockManagerRemoved: SparkListenerBlockManagerRemoved =>  
 listener.onBlockManagerRemoved(blockManagerRemoved)  
 **case** unpersistRDD: SparkListenerUnpersistRDD =>  
 listener.onUnpersistRDD(unpersistRDD)  
 **case** applicationStart: SparkListenerApplicationStart =>  
 listener.onApplicationStart(applicationStart)  
 **case** applicationEnd: SparkListenerApplicationEnd =>  
 listener.onApplicationEnd(applicationEnd)  
 **case** metricsUpdate: SparkListenerExecutorMetricsUpdate =>  
 listener.onExecutorMetricsUpdate(metricsUpdate)  
 **case** executorAdded: SparkListenerExecutorAdded =>  
 listener.onExecutorAdded(executorAdded)  
 **case** executorRemoved: SparkListenerExecutorRemoved =>  
 listener.onExecutorRemoved(executorRemoved)  
 **case** blockUpdated: SparkListenerBlockUpdated =>  
 listener.onBlockUpdated(blockUpdated)  
 **case** logStart: SparkListenerLogStart => *// ignore event log metadata* }  
 }  
}

**private**[spark] **class** StreamingListenerBus  
 **extends** AsynchronousListenerBus[StreamingListener, StreamingListenerEvent](**"StreamingListenerBus"**)  
 **with** Logging {  
  
 **private val** *logDroppedEvent* = **new** AtomicBoolean(**false**)  
  
 **override def** onPostEvent(listener: StreamingListener, event: StreamingListenerEvent): Unit= {  
 event **match** {  
 **case** receiverStarted: StreamingListenerReceiverStarted =>  
 listener.onReceiverStarted(receiverStarted)  
 **case** receiverError: StreamingListenerReceiverError =>  
 listener.onReceiverError(receiverError)  
 **case** receiverStopped: StreamingListenerReceiverStopped =>  
 listener.onReceiverStopped(receiverStopped)  
 **case** batchSubmitted: StreamingListenerBatchSubmitted =>  
 listener.onBatchSubmitted(batchSubmitted)  
 **case** batchStarted: StreamingListenerBatchStarted =>  
 listener.onBatchStarted(batchStarted)  
 **case** batchCompleted: StreamingListenerBatchCompleted =>  
 listener.onBatchCompleted(batchCompleted)  
 **case** outputOperationStarted: StreamingListenerOutputOperationStarted =>  
 listener.onOutputOperationStarted(outputOperationStarted)  
 **case** outputOperationCompleted: StreamingListenerOutputOperationCompleted =>  
 listener.onOutputOperationCompleted(outputOperationCompleted)  
 **case** \_ =>  
 }  
 }  
  
 **override def** onDropEvent(event: StreamingListenerEvent): Unit = {  
 **if** (*logDroppedEvent*.compareAndSet(**false**, **true**)) {  
 *// Only log the following message once to avoid duplicated annoying logs.* logError(**"Dropping StreamingListenerEvent because no remaining room in event queue. "** +  
 **"This likely means one of the StreamingListeners is too slow and cannot keep up with the "** +  
 **"rate at which events are being started by the scheduler."**)  
 }  
 }  
}

D),消息的缓存

消息是缓存在AsynchronousListenerBus

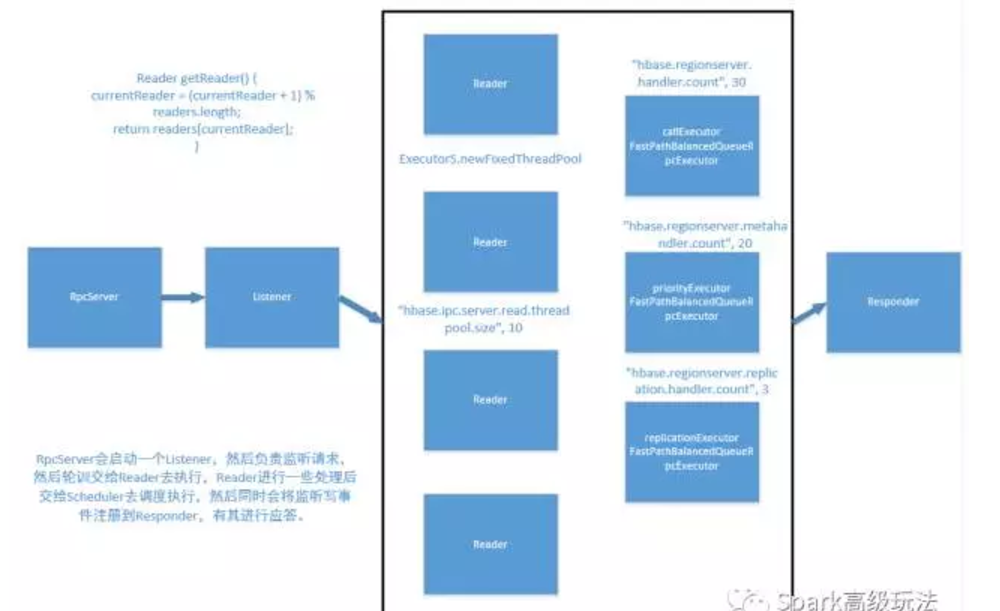
**val** *eventQueue* = **new** LinkedBlockingQueue[E](EVENT\_QUEUE\_CAPACITY)

EVENT\_QUEUE\_CAPACITY=10000

**三，Spark多进程之间的通讯RPC**

Spark的内部rpc老版本是用akka实现的，spark1.6以后虽然保留akka，但是默认实现已经是netty。

其实，rpc采用netty之前，rpc是通过akka，而文件传输是通过netty。现在相当于全部采用了netty的实现的。



**四，总结**

本篇文章主要是将内部spark的内部事件通知的机制。希望通过这篇文章，大家对spark内部事件通知流程有所了解。

这三种模型是我们现在编程最常见的三种模型，希望能对大家编写自己的代码提供一些有益的思路。